**Final Year B. Tech., Sem VII 2024-25**

High Performance Computing Lab

**Practical No. 9**

PRN : 21510111

Name : Siddharth Salunkhe

# Implement Matrix-Vector Multiplication using MPI. Use different number of processes and analyze the performance.

Code :

*#include* <mpi.h>

*#include* <stdio.h> *#include* <stdlib.h> *#include* <time.h>

*#define* N 4 *// Matrix size (NxN)*

void matrix\_vector\_multiply(int *rank*, int *size*, int *n*, double\*

*A*, double\* *x*, double\* *local\_result*) {

int local\_rows = *n* / *size*; *// Number of rows for each process*

*for* (int i = 0; i < local\_rows; i++) {

*local\_result*[i] = 0;

*for* (int j = 0; j < *n*; j++) {

*local\_result*[i] += *A*[i \* *n* + j] \* *x*[j];

}

}

}

*// Sequential version of matrix-vector multiplication for single-process time*

void sequential\_matrix\_vector\_multiply(int *n*, double\* *A*, double\*

*x*, double\* *result*) {

*for* (int i = 0; i < *n*; i++) {

*result*[i] = 0;

*for* (int j = 0; j < *n*; j++) {

*result*[i] += *A*[i \* *n* + j] \* *x*[j];

}

}

}

int main(int *argc*, char\*\* *argv*) { int rank, size;

int n = N; *// Matrix size*

double \*A = NULL, \*x = NULL, \*result = NULL; double \*local\_A, \*local\_result;

int local\_rows;

MPI\_Init(&*argc*, &*argv*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

local\_rows = n / size; *// Number of rows per process*

*// Allocate memory for local matrices and vectors*

local\_A = (double\*) malloc(local\_rows \* n \* sizeof(double)); local\_result = (double\*) malloc(local\_rows \* sizeof(double));

*// Master process initializes the matrix and vector if* (rank == 0) {

A = (double\*) malloc(n \* n \* sizeof(double)); x = (double\*) malloc(n \* sizeof(double));

result = (double\*) malloc(n \* sizeof(double));

*// Initialize matrix A and vector x with random values*

srand(time(0));

*for* (int i = 0; i < n \* n; i++) { A[i] = rand() % 10;

}

*for* (int i = 0; i < n; i++) { x[i] = rand() % 10;

}

printf("Matrix A:\n");

*for* (int i = 0; i < n; i++) {

*for* (int j = 0; j < n; j++) {

printf("%6.2f ", A[i \* n + j]);

}

printf("\n");

}

printf("\nVector x:\n");

*for* (int i = 0; i < n; i++) { printf("%6.2f\n", x[i]);

}

}

*// Sequential execution to calculate T1*

double T1\_start, T1\_end, T1;

*if* (rank == 0) {

T1\_start = MPI\_Wtime(); *// Start timing for sequential execution*

double\* seq\_result = (double\*) malloc(n \* sizeof(double));

sequential\_matrix\_vector\_multiply(n, A, x, seq\_result);

T1\_end = MPI\_Wtime(); *// End timing for sequential execution*

T1 = T1\_end - T1\_start;

printf("\nSequential Result vector:\n");

*for* (int i = 0; i < n; i++) {

printf("%6.2f\n", seq\_result[i]);

}

free(seq\_result);

}

*// Scatter the matrix rows to all processes*

MPI\_Scatter(A, local\_rows \* n, MPI\_DOUBLE, local\_A, local\_rows \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

*// Broadcast the vector to all processes if* (rank == 0) {

MPI\_Bcast(x, n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

} *else* {

x = (double\*) malloc(n \* sizeof(double));

MPI\_Bcast(x, n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

*// Parallel execution to calculate Tp*

double Tp\_start, Tp\_end, Tp;

Tp\_start = MPI\_Wtime(); *// Start timing for parallel execution*

*// Perform local matrix-vector multiplication*

matrix\_vector\_multiply(rank, size, n, local\_A, x, local\_result);

*// Gather the local results into the global result*

MPI\_Gather(local\_result, local\_rows, MPI\_DOUBLE, result, local\_rows, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

Tp\_end = MPI\_Wtime(); *// End timing for parallel execution*

Tp = Tp\_end - Tp\_start;

*// Master process prints the parallel result and calculates speedup*

*if* (rank == 0) {

printf("\nParallel Result vector:\n");

*for* (int i = 0; i < n; i++) {

printf("%6.2f\n", result[i]);

}

*// Calculate speedup: Speedup = T1 / Tp*

double speedup = T1 / Tp;

printf("\nTime (Sequential - T1): %f seconds\n", T1); printf("Time (Parallel - Tp): %f seconds\n", Tp);

printf("Speedup: %f\n", speedup);

}

*// Clean up*

free(local\_A);

free(local\_result);

*if* (rank == 0) { free(A);

free(x);

free(result);

} *else* {

free(x);

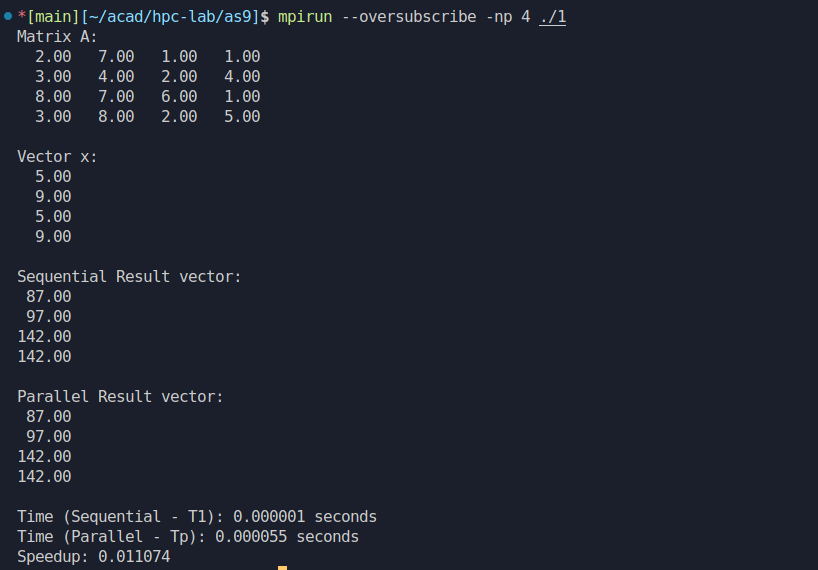
}

MPI\_Finalize();

*return* 0;

}

Output :



# Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.

Code :

*#include* <mpi.h>

*#include* <stdio.h> *#include* <stdlib.h> *#include* <time.h>

*#define* N 4 *// Matrix size (NxN)*

void matrix\_multiply(int *rank*, int *size*, int *n*, double\* *A*, double\* *B*, double\* *local\_C*) {

int local\_rows = *n* / *size*; *// Number of rows assigned to each process*

*for* (int i = 0; i < local\_rows; i++) {

*for* (int j = 0; j < *n*; j++) {

j];

*local\_C*[i \* *n* + j] = 0;

*for* (int k = 0; k < *n*; k++) {

*local\_C*[i \* *n* + j] += *A*[i \* *n* + k] \* *B*[k \* *n* +

}

}

}

}

*// Sequential version for time comparison*

void sequential\_matrix\_multiply(int *n*, double\* *A*, double\* *B*, double\* *C*) {

*for* (int i = 0; i < *n*; i++) {

*for* (int j = 0; j < *n*; j++) {

*C*[i \* *n* + j] = 0;

*for* (int k = 0; k < *n*; k++) {

*C*[i \* *n* + j] += *A*[i \* *n* + k] \* *B*[k \* *n* + j];

}

}

}

}

int main(int *argc*, char\*\* *argv*) { int rank, size;

int n = N; *// Matrix size*

double \*A = NULL, \*B = NULL, \*C = NULL; double \*local\_A, \*local\_C;

int local\_rows;

MPI\_Init(&*argc*, &*argv*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

local\_rows = n / size; *// Number of rows per process*

*// Allocate memory for local matrices*

local\_A = (double\*) malloc(local\_rows \* n \* sizeof(double)); local\_C = (double\*) malloc(local\_rows \* n \* sizeof(double));

*// Master process initializes matrices A and B if* (rank == 0) {

A = (double\*) malloc(n \* n \* sizeof(double)); B = (double\*) malloc(n \* n \* sizeof(double)); C = (double\*) malloc(n \* n \* sizeof(double));

*// Initialize matrix A and B with random values*

srand(time(0));

*for* (int i = 0; i < n \* n; i++) { A[i] = rand() % 10;

B[i] = rand() % 10;

}

printf("Matrix A:\n");

*for* (int i = 0; i < n; i++) {

*for* (int j = 0; j < n; j++) {

printf("%6.2f ", A[i \* n + j]);

}

printf("\n");

}

printf("\nMatrix B:\n");

*for* (int i = 0; i < n; i++) {

*for* (int j = 0; j < n; j++) {

printf("%6.2f ", B[i \* n + j]);

}

printf("\n");

}

}

*// Broadcast matrix B to all processes if* (rank == 0) {

MPI\_Bcast(B, n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

} *else* {

B = (double\*) malloc(n \* n \* sizeof(double));

MPI\_Bcast(B, n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

*// Scatter rows of matrix A to all processes*

MPI\_Scatter(A, local\_rows \* n, MPI\_DOUBLE, local\_A, local\_rows \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

*// Perform local matrix multiplication*

double start\_time = MPI\_Wtime(); *// Start parallel time*

matrix\_multiply(rank, size, n, local\_A, B, local\_C);

double end\_time = MPI\_Wtime(); *// End parallel time*

*// Gather the results back into matrix C in the master process*

MPI\_Gather(local\_C, local\_rows \* n, MPI\_DOUBLE, C, local\_rows

\* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

*// Master process prints the final matrix C if* (rank == 0) {

printf("\nMatrix C (A \* B):\n");

*for* (int i = 0; i < n; i++) {

*for* (int j = 0; j < n; j++) {

printf("%6.2f ", C[i \* n + j]);

}

printf("\n");

}

printf("\nParallel execution time: %f seconds\n", end\_time - start\_time);

}

*// Sequential execution to calculate T1 for speedup comparison*

*if* (rank == 0) {

double\* seq\_C = (double\*) malloc(n \* n \* sizeof(double));

double seq\_start\_time = MPI\_Wtime();

sequential\_matrix\_multiply(n, A, B, seq\_C); double seq\_end\_time = MPI\_Wtime();

double sequential\_time = seq\_end\_time - seq\_start\_time;

printf("\nSequential execution time: %f seconds\n", sequential\_time);

*// Speedup calculation*

double speedup = sequential\_time / (end\_time - start\_time);

printf("\nSpeedup: %f\n", speedup);

free(seq\_C);

}

*// Clean up*

free(local\_A); free(local\_C);

*if* (rank == 0) { free(A);

free(B);

free(C);

} *else* {

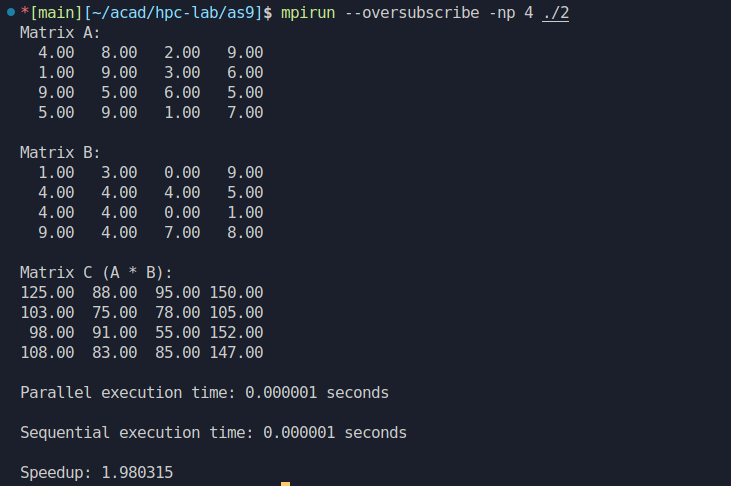
free(B);

}

MPI\_Finalize();

*return* 0;

}

Output :